

CMPT 280

Tutorial: Timing Analysis

Mark G. Eramian

University of Saskatchewan

Common Growth Functions

- From most slowly growing, to most quickly growing, some common growth functions:
 - $\log \log n$
 - $\log n$ (logarithmic)
 - \sqrt{n}
 - n (linear)
 - $n \log n$
 - n^2 (quadratic)
 - n^3 (cubic)
 - ...
 - n^k (polynomial hierarchy)
 - a^n (exponential hierarchy)
 - $n!$

Growth Functions in Big- O Notation

Which of these growth functions belong to $O(n)$? $O(n^2)$? $O(n^3)$? $O(2^n)$? $O(\log n)$?

1. $5 \log n + \sqrt{n} + 1000n^2$
2. $15n \log n + 2^n - 100$
3. $42n^3 + 3n^2 + 2n + 1$
4. $7n + 1400n! + \frac{12}{7} \log n$
5. $7700n^2 \log n$
6. $\frac{8}{11}n + 8\frac{\log n}{2} + 17(n - 1)$

Statement Counting

Example: arrayMax

```
1 Algorithm arrayMax(A, n)
2 Precond: A is an array of n integers.
3 Returns: value of largest element of A
4
5 currentMax <- A[0]
6 i <- 1
7
8 while (i < n)
9     if ( currentMax < A[i] )
10         currentMax <- A[i]
11     i <- i + 1
12
13 return currentMax
```

- Loop Body: *3 or 4 statements*
 - Single loop iteration: *3 or 4 statements*
 - Loop executed $n - 1$ times.
- Best case (if never true): $T_{arrayMax}(n) = 2 + 3(n - 1) + 1 + 1 = 3n + 1$
- Worst case (if always true): $T_{arrayMax}(n) = 2 + 4(n - 1) + 1 + 1 = 4n$

Statement Counting

Example: arrayMax

```
1 Algorithm arrayMax(A, n)
2 Precond: A is an array of n integers.
3 Returns: value of largest element of A
4
5 currentMax <- A[0]
6 i <- 1
7
8 while (i < n)
9     if ( currentMax < A[i] )
10         currentMax <- A[i]
11     i <- i + 1
12
13 return currentMax
```

- Loop Body: *3 or 4 statements*
 - Single loop iteration: *3 or 4 statements*
 - Loop executed $n - 1$ times.
- Best case (if never true): $T_{arrayMax}(n) = 2 + 3(n - 1) + 1 + 1 = 3n + 1$
- Worst case (if always true): $T_{arrayMax}(n) = 2 + 4(n - 1) + 1 + 1 = 4n$

$$T_{arrayMax}(n) \in O(n), T_{arrayMax}(n) \in \Theta(n).$$

Active Operation

Example: arrayMax

```
1 Algorithm arrayMax(A, n)
2 Precond: A is an array of n integers.
3 Returns: value of largest element of A
4
5 currentMax <- A[0]
6 i <- 1
7
8 while (i < n)
9     if ( currentMax < A[i] )
10         currentMax <- A[i]
11     i <- i + 1
12
13 return currentMax
```

- Active operation: `while (i < n)`
- Number of executions of active operation: n

Active Operation

Example: arrayMax

```
1 Algorithm arrayMax(A, n)
2 Precond: A is an array of n integers.
3 Returns: value of largest element of A
4
5 currentMax <- A[0]
6 i <- 1
7
8 while (i < n)
9     if ( currentMax < A[i] )
10         currentMax <- A[i]
11     i <- i + 1
12
13 return currentMax
```

- Active operation: `while (i < n)`
- Number of executions of active operation: n

$$T_{arrayMax(n)} \in O(n), T_{arrayMax(n)} \in \Theta(n).$$

Common Analysis Cases

Linear Loops

Simple counting loops are Linear Loops:

```
1 for(i = 0; i < n; i++) {  
2     <statements>  
3 }  
4  
5 i = 0;  
6 while(i < n) {  
7     <statements>  
8     i++  
9 }
```

Loop executes n times (n is size of input) As long as number of statements in loop body is independent of n , such a loop is $\Theta(n)$.

This loop is also linear. Why?

```
1 for (i = 0; i < n; i+=2)  
2     <statements>  
3 end for
```


Common Analysis Cases

Logarithmic Loops

Logarithmic Loops result when the counter is multiplied or divided each iteration:

```
1 for(i = 1; i < n; i = i*2)
2     <loop body>
3 end for
4
5 for(i = n; i >= 1; i = i/2)
6     <loop body>
7 end for
```

Claim: Each of these loops executes $f(n) = c \log_2(n)$ times where c is the number of statements in the loop body. Thus each loop is $\Theta(\log n)$.

Common Analysis Cases

Logarithmic Loops

```
1 for(i = 1; i < n; i = i*2)
2   <loop body>
3 end for
```

Consider the value of i in the above loop:

- First iteration: $i = 1 = 2^0$
- Second iteration: $i = 2 = 2^1$
- Third iteration: $i = 4 = 2^2$
- j -th iteration: $i = 2^{j-1}$

Solving the last equation for j reveals that $j = \log i + 1$. Thus, if the loop stops on the j -th iteration, we know that $i \geq n$. But we also know that on the $j - 1$ -th iteration, $i < n$. Thus, i can only be slightly bigger than n at most. Substituting this into $j = \log i + 1$ we obtain $j = \log n + 1$, but since j must be an integer, we have round up: $j = \lceil \log n + 1 \rceil$. This is the maximum possible value for j , thus the loop executes $O(\log n)$ times.

Common Analysis Cases

Quadratic Nested Loops

Simple quadratic loops occur when the inner and outer loops each execute the same number of times:

```
1 for(i = 0; i < n; i++)  
2     for(j = 0; j < n; j++)  
3         <loop body containing c statements>  
4     end loop  
5 end loop
```

Thus, total number of statements is $f(n) = c \times n \times n$ which is $\Theta(n^2)$ (Assuming no methods in the loop body with time $> O(1)$).

What if the inner loop was `for(j = 0; j < m; j++)`?

Common Analysis Cases

Dependent Quadratic Nested Loops

Dependent quadratic loops result when the number of iterations in the inner loop depends on the value of the outer loop counter:

```
1 for(i = 0; i < n; i++)  
2   for(j = 0; j < i; j++)  
3     <loop body containing c statements>  
4   end loop  
5 end loop
```

Number of loop body statements:

$$\begin{aligned} 0 + c + 2c + 3c + 4c + \dots + (n-1)c &= c \cdot (1 + 2 + 3 + \dots + n-1) \\ &= c \cdot \sum_{i=1}^{n-1} i \\ &= c \cdot \frac{(n-1)n}{2} \\ &= \frac{c}{2}(n^2 - n) \in \Theta(n^2) \end{aligned}$$

Example: Matrix Sum

```
1 Algorithm addMatrix( matrix1, matrix2, matrix3, n)
2 Precond: matrix1 and matrix2 are 2D arrays of numbers of size n by n
3 Postcond: matrix3 contains the sum of matrix1 and matrix2
4
5 for(i = 0; i < n; i++) {
6     for(j = 0; j < n; j++) {
7         matrix3[i][j] = matrix1[i][j] + matrix2[i][j]
8     }
9 }
```

What is the time complexity in the worst case? Best case?

Example: Prefix Averages

```
1 Algorithm prefixAverages(X, n)
2 Precond: X is an n-element array numbers
3 Output: An n-element array A of numbers such that A[i]
4         is the average of X[0] : X[i]
5
6 i = 0
7 while (i < n) {
8     avg = 0
9     j = 0
10    while ( j <= i ) {
11        avg = avg + X[j]
12        j++
13    }
14    A[i] = avg/(i+1)
15    i++
16 }
17 return A
```

What is the time complexity in the worst case? Best case?

Example: Binary Search

```
1 Algorithm binarySearch (arr, n, key)
2 Pre: arr is a sorted (ascending order) integer array of length n
3     key is value for which to search in arr
4 Post: arr is unchanged
5 Return: index of position of key in arr, -1 if not found
6
7 lo = 0
8 hi = n-1
9 while ( lo <= hi )
10     mid = (lo + hi) / 2
11     if( key < arr[mid] )
12         hi = mid - 1
13     else if ( key > arr[mid] )
14         lo = mid + 1
15     else
16         return mid;
17 return -1;
```

What's the time complexity of this algorithm in the worst case?
Best case?